Design Rationale

Romal Patel

Classes

A key class that I have included in my diagram is the DragonCard class which resembles the Dragon Cards from the game and has an association with the DragonCardPool class. The basic outline of the relationship between the two classes is that the DragonCardPool class is a panel that contains all the unique DragonCard classes that the player can choose from. Initially the idea was to contain all the unique Dragon Cards within the DragonCardPool class as ArrayList attribute but that would have introduced difficulties in handling the cards in terms of their actions/effects and create further problems of extensibility. Having a class instead allows for further methods to be added in the future if extensions are required or thought of. It also allows for the Card to hold certain attributes like the images for the card, name and quantity of the action. Overall, it makes the creation as well as the use of the card's features much more accessible and structured.

Another key class that I have decided to add is the Creature class. At first, the idea was to have an enumeration of each creature. However, I decided against this and created a class instead as I wanted to store additional information on each type of creature. For example, each type of Creature has its own image and name which make it simple when implementing a creature onto a card. It is also able to distinguish whether it is a pirate card or not. This is much more effective than creating unnecessary methods that are used repeatedly to match names to image files or check if the creature is a pirate or not. Furthermore, it creates room for extensibility and creativity for further improvements in the game. For example, by having a specific class for each creature the developer is able to add creature specific methods that give more functionality and features to the game. Overall, the creature class acts as an information holder class that other classes can utilise to quickly access files and information on the creature. For future sprints, it will evolve and become more useful when it comes to adding additional features and functions.

Relationships

A key relationship that is displayed on the UML class diagram is the aggregation relationship between Board and DragonCardPool classes. The reason I have chosen an aggregation relationship here is because there is a strong relationship between the two classes, where it could be said that the Board class requires the DragonCardPool class to function correctly and coherently. However, the DragonCardPool does not always require the Board class, it can exist independently and be placed on any other frame or panel. Hence, I described the relationship as an aggregation. However, it can be argued that in this scenario, specific to the context of the game and its requirements, that the relationship can be a composite relationship instead as the DragonCardPool should only really exist on the Board class. The fact that Board controls the instantiation of the DragonCardPool class and also exclusively, for the

most part, manages the class hints at a composite relationship. Additionally, the life span of the two classes coincide, adding to the argument of a composite relationship. Therefore, depending on how you view the situation, there are arguments for both composite and aggregation.

Moving on, an interesting relationship is the relationship between DragonToken and Cave as there can be an argument between dependency and association. As per the UML class diagram, I have chosen to go with a dependency relationship as by definition, a dependency occurs when one class uses another only during an execution of a method, or in other words, only depends on the class for temporarily to execute something. In this case, when a DragonToken needs to be placed in a Cave, there is only a temporary establishment between the two classes where Cave places the Token on its panel's location. Other than this, they should not have much of a relationship and thus be independent of each other. This pushes for a more flexible and decoupled design. However, I believe that eventually this design may need to be altered to an association, where DragonToken will need to hold reference to Cave in order to remember which Cave it belongs to and, hence, should return to. However, I am not entirely sure about this route, and therefore have left it as a dependency for now. This design may need to be altered in a future Sprint if a situation arises.

Inheritance

The design includes an inheritance call from Creature to each of the types of creature (Bat, Salamander, Spider, BabyDragon and PirateDragon). The idea behind the Creature class as an abstract class for all creatures has been pointed out in previous statements. The main idea around inheritance in this case, is for the children to inherit the properties of the Creature class. As such, each creature has a name, and image, and a status of whether it is an enemy/pirate or not. Although I have not made this clear in the UML class diagram, this is a similar case for the abstract Cave class where the different types of caves inherit the properties and methods of the Cave class. This approach allows for a modular and scalable design. It allows for each subclass to extend functionality, yet make sure common attributes are uniformly handled and not cluttered through various lines of code.

Cardinalities

In terms of cardinalities, the first one I would like to point out is the cardinality between Board and DragonToken. From Board to DragonToken, the cardinality is 2..4. I have chosen this instead of, for example, 0..4 or 1..4 due to the fact that the game cannot be played with 0 or 1 players. Atleast, this is the case till a bot or computer-automated player is introduced to the game. The max DragonTokens is 4, instead of more, as there are only 4 caves at the moment, outlined by the cardinality from Board to Cave. Each DragonToken needs to belong to a Cave, hence, until there are no additional caves added, no more DragonTokens can be introduced to the game. Another cardinality set worth mentioning is from VolcanoCard to Cave, which is 0..1. The idea behind this is that some Volcano Cards are linked to a cave, whereas some aren’t. Hence, in theory, the attribute for Volcano Card should hold the Cave that it is linked to, otherwise it should be null. Implying that the cardinality is 0..1 and not just 1 or 0..2 for example.

Design Patterns

A design pattern I have incorporated into my design is the singleton class design pattern which restricts multiple instances of a class by storing it as an private static attribute where unless it is null, it cannot be instantiated. This is done by adding a private constructor that cannot be accessed outside the class. Instead, there is a public method that is used to return the instance if it exists already, or create it if not. This design pattern has been implemented for the Board class. The reason for this by context is that there should only ever exist one board at a time. In terms of the game implementation, it prevents inconsistencies in the game state, where multiple instances of Board may cause conflicts. Furthermore, it creates a global point of access, making it easy and efficient to access from anywhere in the code instead of tediously putting it through as a parameter, creating unnecessary complexity within the application. To add, it also gives the Board a role of central management where changes and updates can simply be reflected on to the application. Following the previous point, although this is a niche advantage, and may not apply to this project at this current scope, the advantage of integrating with the Observer design pattern is worth mentioning in case of future expansions of the game. Having only one instance makes it highly efficient to integrate with the observer design pattern in a situation where all users/subscribers need to be notified of changes to the game.